Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Automatic Determination of May/Must Set Usage in Data-Flow Analysis

Andrew Stone
Colorado State University
stonea@cs.colostate.edu

Michelle Strout
Colorado State University
mstrout@cs.colostate.edu

Shweta Behere
Avaya Inc.
behere@avaya.com

Abstract

Data-flow analysis is a common technique to gather program information for use in transformations such as register allocation, dead-code elimination, common sub-expression elimination, scheduling, and others. Tools for generating data-flow analysis implementations remove the need for implementers to explicitly write code that iterates over statements in a program, but still require them to implement details regarding the effects of aliasing, side effects, arrays, and user-defined structures. This paper presents the DFAGen Tool, which generates implementations for locally separable (e.g. bit-vector) data-flow analyses that are pointer, side-effect, and aggregate cognizant from an analysis specification that assumes only scalars. Analysis specifications are typically seven lines long and similar to those in standard compiler textbooks. The main contribution of this work is the automatic determination of may and must set usage within automatically generated data-flow analysis implementations.

1. Introduction

Program analysis is the process of gathering information about programs to effectively derive a static approximation of behavior. This information can be used to optimize programs, aid debugging, verify behavior, and detect potential parallelism.

Data-flow analysis is a common technique for statically analyzing programs. It works by propagating program information, encoded as data-flow values, through a control flow graph of statements or basic blocks. One common example, reaching definitions, propagates sets of statements whose definition may reach later uses in the control-flow graph. Compilers textbooks [1, 3, 4] specify data-flow analyses with data-flow equations. Figure 1 shows a specification of reaching definitions that uses such equations. Each statement has an associated in and out data-flow set. A solution to a data-flow analysis problem is an assignment of data-flow values in all in and out, such that these sets satisfy the equations.

Data-flow analysis problems may be formalized within a lattice-theoretic framework [1, 8], which separates the analysis specification into a transfer function, meet operator, and analysis direction. The transfer functions that can iteratively solve the data-flow equations in Figure 1 computes the out set using the statement-specific gen and $kill$ information and the in set. The meet operator for reaching definitions is set union and is used to calculate the in set for a statement as the union of all out sets for statements that precede the current statement in the control flow. Reaching definitions is a forward analysis; for backward analyses, the role of the in and out sets switch in relation to the transfer function and meet operation.

$$in[s] = \bigcup_{p \in pred[s]} out\,[p]$$

$$out\,[s] = gen\,[s] \cup (in\,[s] - kill\,[s])$$

$$gen[s] = s\text{ , if }\ def[s] \neq \emptyset$$

$$kill[s] = \text{all } t \text{ such that } def[t] \subseteq def[s]$$

Figure 1. Data-flow equations for reaching definitions, where s, p, and t are program statements, $in[s]$ and $out[s]$ are the data-flow sets for statement s, and $def[s]$ is the set of variables assigned at statement s.

Lattice-theoretic frameworks enable a separation of concerns between the logic for a specific data-flow analysis and the iterative algorithm and proof of convergence, which can be generalized to all data-flow analysis problems where the transfer function satisfies certain properties. This separation of concerns has led to the development of a number of data-flow analysis frameworks that ease the implementation of data-flow analyses.

However, complications in specifying data-flow analyses still exist, particularly because of the may versus must nature of variable access in a static analysis. This may versus

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.28

Table 1. Reaching definition data-flow sets for example in Figure 2

Statement , s	$in[s]$	$out[s]$	$gen[s]$	$kill[s]$
S1	{}	{ S1 }	{ S1 }	{ S8, S8 }
S2	{ S1 }	{ S1, S2 }	{ S2 }	{ S2 }
S3	{ S1, S2 }	{ S1, S2, S3 }	{ S3 }	{ S3 }
S4	{S1, S2, S3}	{ S1, S2, S3 }	{ }	{}
S5	{S1, S2, S3}	{S1, S2, S3, S5}	{ S5 }	{S5, S6 }
S6	{S1, S2, S3}	{S1, S2, S3, S6}	{ S6 }	{ S5,S6 }
S7	{S1, S2, S3,S5,S6}	{ S1, S2, S3, S5,S6, S7 }	{ S7 }	{ }
S8	{ S1, S2, S3, S5, S6, S7}	{ S2, S3, S5, S6, S7, S8 }	{ S8 }	{ S1, S8 }
S9	{S2, S3, S5, S6, S7, S8}	{ S2, S3, S5, S6, S7, S8 }	{ }	{ }

```
    int *pointsToOne;
    int *pointsToTwo;
    int a, b;
S1  a = ...
S2  b = ...
S3  pointsToOne = &a;
S4  if(a < b) {
S5     pointsToTwo = &a;
    } else {
S6     pointsToTwo = &b;
    }
S7  *pointsToTwo = ...;
S8  *pointsToOne = ...;
S9  printf("Vals = %d, %d\n",
       *pointsToOne,  *pointsToTwo);
```

Figure 2. May/must definition example.

$$in[s] = \bigcup_{p \in pred[s]} out\,[p]$$

$$out\,[s] = gen\,[s] \cup (in\,[s] - kill\,[s])$$

$$gen[s] = s\text{ , if }\ maydef[s] \neq \emptyset$$

$$kill[s] = \text{all } t \text{ such that } maydef[t] \subseteq mustdef[s]$$

Figure 3. Data-flow equations for reaching definitions that are cognizant of may and must definitions due to aliasing, side-effects, and/or aggregates

must nature occurs due to aliasing, side-effects, and aggregates. Although data-flow equations are typically presented in the context of scalar variables, these other common programming language features must be dealt with when implementing the transfer function for an analysis. For example, in Figure 2 the pointer variable `pointsToOne` *must* point to the variable `a` after statement `S3`. The pointer variable `pointsToTwo` *may* point to variable `a` or `b` in statement `S7`. Therefore statement `S7` is included in the set of reaching definitions generated, but it is unable to kill the statements `S1` and `S2`. Since `pointsToOne` must point at variable `a`, it kills the definition of variable `a` in statement `S1`.

The existence of may and must definitions and uses due to aliasing, side-effects, and aggregates makes it necessary to define transfer functions so that they use may and must sets in a conservatively correct fashion. Since reaching definitions is a may analysis, the set of generated definitions $gen[s]$ should be conservatively large and the set of killed definitions $kill[s]$ should be conservatively small. This requires a more precise definition of the $gen[s]$ and $kill[s]$ sets as shown in Figure 3 in terms of may and must definition sets. Notice however, that even for this common and relatively simple data-flow analysis the assignment of may and must to the definition sets is non-trivial.

This paper describes how the transfer function implementations are automatically generated from succinct descriptions that do not indicate whether sets should be may or must, and that can maintain the "data-flow analysis for scalars" abstraction. DFAGen handles the may/must issues due to aliasing, side-effects, and aggregates by applying a new analysis that determines whether sets should refer to may or must variants based on data-flow equations and analysis style (i.e., may or must). DFAGen also enables reuse between analysis implementations and extensibility through its use of pre-defined sets, which are used to hide code specific to any particular compiler and/or analysis infrastructure.

Section 2 presents how a user specifies a data-flow analysis to the DFAGen Tool. Sections 3.1 through 3.3 describe the phases of the DFAGen Tool, which include data-flow set type inference, may and must determination, and code generation. Section 4 provides experimental results that indicate the data-flow analysis implementations generated by the DFAGen Tool are comparable in terms of performance

```
predefined: defs[s]
    description: Set of variables defined at a given statement.
    argument:    stmt s
    calculates:  set of var, predefined, mStmt2MayDefMap, mStmt2MustDefMap
    maycode:
        // 17 lines of C++ code using OpenAnalysis framework
        // generating mStmt2MayDefMap
    mustcode:
        // 17 lines of C++ code using OpenAnalysis framework
        // generating mStmt2MustDefMap
    end
```

Figure 5. Code specification for def[s] pre-defined set

```
Analysis: ReachingDefinitions
  meet: union
  flowvalue: stmt
  direction: forward
  style: may
  gen{s}:
    {s | defs[s] != empty}
  kill{s}:
    {t | defs[t] subset defs{s}}
```

Figure 6. DFAGen specification for reaching definitions

$$
\begin{array}{rcl}
AnalysisDef & \Rightarrow & \textbf{Analysis} : id \\
 & & \quad \textbf{meet} : \\
 & & \quad\quad (\textbf{union} \,|\, \textbf{intersection}) \\
 & & \quad \textbf{flowtype} : id \\
 & & \quad \textbf{direction} : \\
 & & \quad\quad (\textbf{forward} \,|\, \textbf{backward}) \\
 & & \quad \textbf{style} : (\textbf{may} \,|\, \textbf{must}) \\
 & & \quad \textbf{gen}[\,id\,] : Set \\
 & & \quad \textbf{kill}[\,id\,] : Set \\
Set & \Rightarrow & \textbf{id}[\textbf{id}] \,|\, BuildSet \,|\, emptySet \,|\, Expr \\
Expr & \Rightarrow & Expr \;\; Op \;\; Expr \,|\, Set \\
Cond & \Rightarrow & Expr \;\; CondOp \;\; Cond \,|\, Expr \\
Op & \Rightarrow & \textbf{union} \,|\, \textbf{intersection} \,|\, \textbf{difference} \,| \\
CondOp & \Rightarrow & \textbf{and} \,|\, \textbf{or} \,|\, \textbf{subset} \,|\, \textbf{superset} \,| \\
 & & \textbf{equal} \,|\, \textbf{not equal} \,|\, \textbf{proper subset} \,| \\
 & & \textbf{proper superset} \\
BuildSet & \Rightarrow & \{id : Cond\}
\end{array}
$$

Figure 7. Grammar for analysis, $gen[s]$, and $kill[s]$ set definition

$$
\begin{array}{l}
PredefinedSetDef \quad \Rightarrow \\
\quad \textbf{predefined} : id[\,id\,] \\
\quad\quad description : line \\
\quad\quad \textbf{argument} : id\,id \\
\quad\quad \textbf{calculates} : \\
\quad\quad\quad (id \,|\, \textbf{set of}\; id)\,,\; \textbf{predefined},\; id,\; id \\
\quad \textbf{maycode} : \\
\quad\quad code \\
\quad \textbf{end} \\
\quad \textbf{mustcode} : \\
\quad\quad code \\
\quad \textbf{end}
\end{array}
$$

Figure 8. Grammar for pre-defined set definition

2.4. Pre-defined data-flow sets

Pre-defined sets map program entities such as statements to sets of other program entities that can be used in data-flow equations. Pre-defined sets have may and must implementations associated with them and DFAGen determines which implementation is appropriate for a given context. DFAGen has some built-in pre-defined sets, such as 'variables defined at statement', 'variables used at statement', and 'expressions contained in statement', and provides a means for analysis writers to define their own. Our approach provides enough flexibility to generate new analyses easily.

Figure 5 shows an example pre-defined set specification, in this case for the set of variables defined at a statement.

3. Compilation Phases in DFAGen Tool

The DFAGen Tool automatically compiles the data-flow analysis specifications provided by the user into a data-flow analysis implementation. The current prototype generates code that interoperates with the OpenAnalysis toolkit. During the compilation process, the may/must status of each

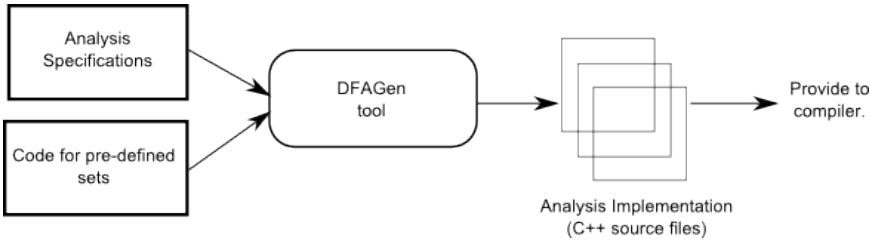

Figure 4. Analysis generation process using the DFAGen Tool

to hand-written versions. The final sections describe related work and conclusions.

2. Using the DFAGen Tool

DFAGen generates analysis implementations for the class of unidirectional, locally separable data-flow problems, which are also referred to as GEN/KILL problems or classical bit-vector problems. This section elaborates on this class of data-flow problems, describes how the DFAGen Tool is used within the context of a compiler infrastructure and analysis framework, presents the DFAGen specification language, and illustrates how pre-defined sets enable extensibility and reuse between analysis specifications.

2.1. Unidirectional, locally separable data-flow problems

DFAGen generates analysis implementations for the class of unidirectional, locally separable data-flow problems. Locally separable (LS) data-flow problems are a subset of finite, distributive, and subset-based (FDS) data-flow problems [10]. The lattice is finite, the transfer functions are distributive, and data-flow facts are subsets of some universal set of finite size. The main difference between LS and FDS is that locally separable data-flow problems assume a transfer function of the form: $f(X) = gen[s] \cup (X - kill[s])$, where $gen[s]$ is the set of data-flow facts generated by statement s, $kill[s]$ is the set of data-flow facts killed by statement s, X is the in/out set for a forward/backward analysis, and $f(X)$ is the out/in set for a forward/backward analysis. The term 'locally separable' indicates that the $gen[s]$ and $kill[s]$ sets do not depend on the $in[s]$ or $out[s]$ sets.

Common examples of locally separable problems are liveness, reaching definitions and available expressions. These analyses as well as others that fall within this category are used in various programs optimizations as well as debugging tools. For example, liveness can be used for dead code elimination and register allocation; reaching definitions can be used for detecting uninitialized variables, loop invariant code motion, generating a program dependence graph [5]; available expressions can be used for common sub-expression elimination etc. Most of these analyses can be also be used in program slicers and debugging tools. Other program optimization examples are busy-code motion, partial dead-code elimination, assignment motion, and strength reduction [1].

2.2. Using DFAGen within a compiler infrastructure

The DFAGen Tool currently generates analysis implementations for compilation and use within the OpenAnalysis framework [11]. Figure 4 illustrates the process by which an analysis specification along with code for calculating the may and must versions of each predefined set are fed into the DFAGen and an analysis implementation is generated. The pieces of the system that are compiler-specific are the code snippets for the predefined sets and the code generation phase of the DFAGen Tool. All other phases are independent and can be used within other compiler infrastructures.

2.3. DFAGen specification language

DFAGen allows compiler writers to specify data-flow analyses with set equations and a small set of properties. The properties to be specified are the meet function, data-flow set type, direction, and analysis style (may or must). Figure 6 shows the DFAGen specification for reaching definitions. Each property is specified with a simple keyword, for example, the meet function for reaching definitions is specified with the '`union`'. The $gen[s]$ and $kill[s]$ sets that define the transfer function are specified in terms of mathematical set notation. These equations can reference predefined sets, such as `defs[s]`, which is the set of definitions generated at statement `s`. They can also use set operations such as union, intersection, and difference, and use set construction to build sets based on various conditional operations such as subset, proper subset, and set equality.

Figure 7 shows the grammar for specifying $gen[s]$ and $kill[s]$ sets in the DFAGen specification language.

pre-defined set in the analysis specification is inferred automatically. Prior to performing may/must analysis, DFAGen parses the specification and predefined set files, constructs an abstract syntax tree for the specification, and verifies analysis legality. After may/must analysis, the DFAGen Tool generates code that implements the specified analysis.

3.1. Type Inference and Type Checking

Type Inference Prior to generating code it is necessary to ensure that the specified data-flow equations use types consistently. For example, the type for GEN and KILL sets are inferred and checked against the specified flow-type. DFAGen's first phase of compilation generates Abstract Syntax Trees (ASTs) for the whole specification including the GEN and KILL equations. Type inference analyzes these ASTs in a bottom-up manner assigning type information to each AST node.

We can directly infer the types for pre-defined set reference nodes from their specifications. All leaf nodes in the AST are guaranteed to be references to either pre-defined sets or the empty set. From leaf node types we can synthesize parent types.

Another motivation for type inference is to determine the domain of values for which the condition in a set builder should be checked. Figure 7 presents the syntax for set builders, and Figure 6 shows examples of its usage.

Types the current DFAGen prototype can handle include statements, expressions, variables, and sets of these types.

Type Checking After type inference DFAGen performs type checking. For any set operator, the type of its operands should match. Also, $gen[s]$ and $kill[s]$ sets should be typed as the set of the specified flow value.

Example: Reaching Definitions Figure 9 presents the results of applying type inference on the example in Figure 6. The type for predefined set references can be inferred from their specification. For example, defs is specified to be type 'set of vars' in Figure 5. The return type for conditional operators is always 'boolean'. We infer that the BuildSet nodes are typed as 'set of stmts' from the flow-type specified in Figure 6.

3.2. Propagating May and Must Information

In DFAGen's specification language the 'style' keyword is used to specify whether an analysis is may or must. A may analysis propagates data-flow values representing execution behavior that may be true. A must analysis propagates data-flow values representing execution behavior that must, under all possible executions, be true. For example, liveness analysis determines, at each statement in a program, the set of variables that may be referenced before redefinition. Since liveness is concerned with what variables may be referenced, rather than those that are guaranteed to be referenced, it is a may analysis.

All pre-defined sets have two variants: may and must. DFAGen determines which variant to use by performing may/must analysis. May/must analysis analyzes $gen[s]$ and $kill[s]$ equation ASTs in a top-down manner tagging nodes as either upper or lower bounded. A node tagged as 'upper' requires its child nodes to be tagged in mannar such that the generated code will produce the largest possible set. Similarly, a node tagged as 'lower' needs child nodes tagged so as to produce the smallest possible set. A pre-defined set reference tagged as 'upper' indicates that generated code will use the may variant of that set, and a pre-defined set reference tagged as 'lower' indicates the must variant will be used.

The may/must analysis assigns the top nodes for the GEN and KILL expressions in the AST upper/lower bound values based on style of the specified analysis (may or must). Table 2 shows how this bound is determined. Table 3 shows how upper and lower bound values are propagated to left and right children for the various set operations currently implemented. The next few sections explain and prove the correctness of the contents of Table 3.

Determining may/must of expression nodes Each of DFAGen's binary expression operators is shown in the Op production of Figure 7. Each of these operators have left and right operands that reference or compute sets, these sets are tagged either as 'upper' or 'lower'. There are four permutations of upper/lower values for two operands. We establish lattices of these permutations. These lattices have unique top and bottom may/must values, which when applied to the op node's children will generate the upper and lower bound sets respectively.

We use the notation that the left side of an operator is either some lower bound set a_l, or some upper bound set a_u, and that the right side is either some lower bound set b_l or some upper bound set b_u. In the next few sections, we establish lattices for the difference, union, and intersection operators, and the subset relation. The lattices are shown graphically in Figure 11. In the following proofs the partial ordering operator (represented as $\leq$) is subset equals unless otherwise indicated.

Difference Given two sets u and l where u is an upper bound set and l is a lower bound set such that $l \leq u$, we know the following relationships hold for any set x:

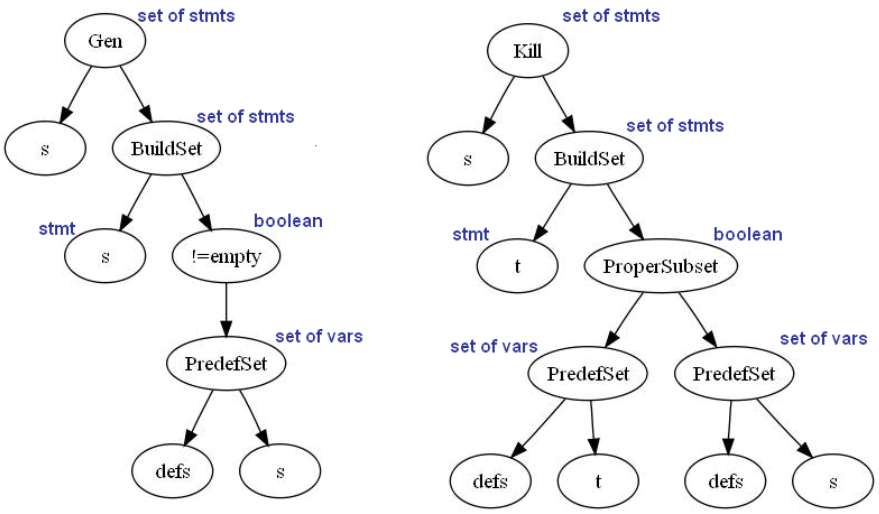

Figure 9. Type checking for reaching definitions

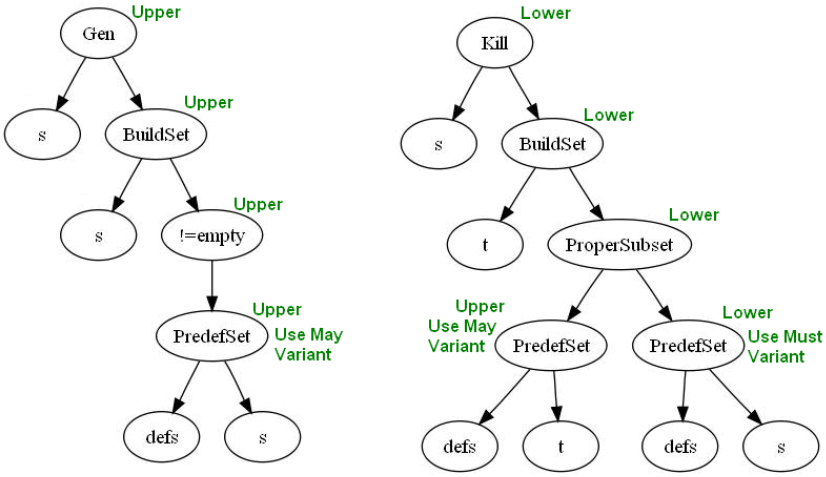

Figure 10. May/must propagation for reaching definitions

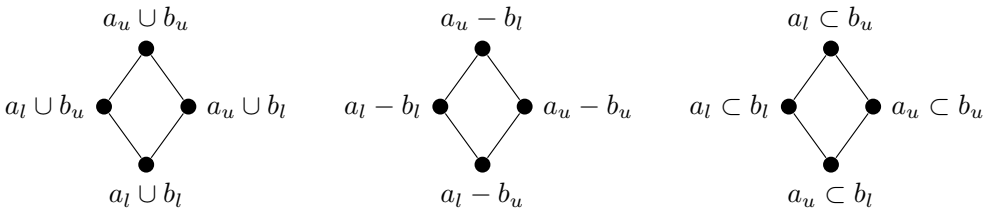

Figure 11. Lattices for difference and union operators, and subset relation

$$x - u \leq x - l \quad (1)$$

$$l - x \leq u - x \quad (2)$$

The left child operand for the difference operator can be a_l or a_u, where $a_l \leq a_u$. A similar relationship holds for the right child operand, $b_l \leq b_u$. Based on those relationships and those in equations (1) and (2), the following partial ordering holds between the four possible operand variants for the difference operator.

$$a_l - b_u \leq a_u - b_u \leq a_u - b_l \quad (3)$$

$$a_l - b_u \leq a_l - b_l \leq a_u - b_l \quad (4)$$

Union and Intersection Given two sets u and l where u is an upper bound set and l is a lower bound set such that $l \leq u$, we know that given any set x:

$$x \cup l \leq x \cup u \quad (5)$$

$$l \cup x \leq u \cup x \quad (6)$$

The same holds true for intersection:

$$x \cap l \leq x \cap u \quad (7)$$

$$l \cap x \leq u \cap x \quad (8)$$

Similar to difference we establish a partial ordering for union and intersection. The ordering for union is:

$$a_l \cup b_l \leq a_l \cup b_u \leq a_u \cup b_u \quad (9)$$

$$a_l \cup b_l \leq a_u \cup b_l \leq a_u \cup b_u \quad (10)$$

The ordering for intersection is the same.

3.2.1 Determining may/must of condition nodes

Condition nodes are used within the context of set-builder expressions. The upper-bound of a set-builder expression occurs when the condition is evaluated as 'true' as many times as possible, the lower-bound occurs when the condition is evaluated as 'false' as many times as possible. All condition operators are shown in the CondOp production of Figure 7.

Similar to the set operation nodes we establish a partial ordering on all possible may/must permutations for the left and right operands of a condition operator. It has been established [1] that in a partial ordering:

$$x \leq y \text{ if and only if } x \wedge y = x \quad (11)$$

for some meet operator $\wedge$ that is idempotent, commutative, and associative. For this proof we use logical-and as the meet operator.

We use the following four facts about the subset relation:

Meet	Style	Gen	Kill
union	may	Upper bound	Lower bound
intersection	must	Lower bound	Upper bound

Table 2. Determining bounds based on meet operator

$$a_u \subset b_l \Rightarrow a_l \subset b_l \quad (12)$$

$$a_u \subset b_u \Rightarrow a_l \subset b_u \quad (13)$$

$$a_l \subset b_l \Rightarrow a_l \subset b_u \quad (14)$$

$$a_u \subset b_l \Rightarrow a_u \subset b_u \quad (15)$$

to establish the following lattice:

$$a_u \subset b_l \leq a_l \subset b_l \leq a_l \subset b_u \quad (16)$$

$$a_u \subset b_l \leq a_u \subset b_u \leq a_l \subset b_u \quad (17)$$

To prove these relations we note that equation (11) holds when $x \wedge y \Leftrightarrow x$. $x \wedge y \Rightarrow x$ is trivially true. To show $x \Rightarrow x \wedge y$, it is sufficient to show $x \Rightarrow y$.

Using equation 11 we develop four implications that prove the relations in the lattice:

$$(a_u \subset b_l) \wedge (a_l \subset b_l) \Longleftrightarrow (a_u \subset b_l) \quad (18)$$

$$(a_l \subset b_l) \wedge (a_l \subset b_u) \Longleftrightarrow (a_l \subset b_l) \quad (19)$$

$$(a_u \subset b_l) \wedge (a_u \subset b_u) \Longleftrightarrow (a_u \subset b_l) \quad (20)$$

$$(a_u \subset b_u) \wedge (a_l \subset b_u) \Longleftrightarrow (a_u \subset b_u) \quad (21)$$

Since it's the case that 11 holds when $x \Rightarrow y$, it is also the case that:

- Equation 18 holds since equation 12 holds.
- Equation 19 holds since equation 14 holds.
- Equation 20 holds since equation 15 holds.
- Equation 21 holds since equation 14 holds.

3.2.2 Example: Reaching Definitions

Consider Figure 10 to understand how may and must propagation takes place for reaching definitions (specified in Figure 6).

As the meet operator is union and the type is may, the $gen[s]$ set node is tagged as 'upper bound' and the $kill[s]$ set node is tagged as 'lower bound'. This information is propagation from top to bottom in the abstract syntax trees. As we reach the pre-defined sets, based on the bound information, we decide whether is a may or must node. The set

	Upper bound		Lower bound	
	lhs	rhs	lhs	rhs
difference	upper	lower	lower	upper
union	upper	upper	lower	lower
intersection	upper	upper	lower	lower
subset	lower	upper	upper	lower
superset	upper	lower	lower	upper
proper subset	lower	upper	upper	lower
proper superset	upper	lower	lower	upper
equal	upper	upper	lower	lower
not equal	upper	upper	lower	lower
not equal to empty set	upper	-	lower	-

Table 3. Determining may/must information

has type may if it has an upper bound and has a type must if it has a lower bound. For the set operator $!=$ empty, the lhs child takes the type may. Thus, the $defs[s]$ for the $gen[s]$ set has type may. A similar process is followed to determine the type of the leaf nodes for the $kill[s]$ set.

3.3 Code Generation

Generated analyses follow an iterative approach for data-flow analysis, and do not use work-lists. The analysis takes previously generated alias analysis results, and a control-flow graph, as parameters. Nodes in the control flow graph are visited and the transfer statement is called for the node only if the IN and OUT sets change for the node. Based on the specified flow-value, we develop data flow sets. Depending on the type of the analysis, we initialize the top and bottom of the lattice. OpenAnalysis has an interface named as DFAGenDFSet for the DFAGen Tool. Various classes implement this interface such as StmtDFSet, ExprDFSet and LocDFSet based on the available types in DFAGen such as stmt, expr and var. Once we know the may/must information for the pre-defined sets, code for the pre-defined sets is directly obtained from their specification file. DFAGen generates code for the $gen[s]$ and $kill[s]$ sets corresponding to their specification. The results of alias analysis and side-effect analysis are utilized in the generation of the analysis specification to obtain precise results. DFAGen can calculate the $gen[s]$ and $kill[s]$ sets in the transfer function or it can pre-calculate these sets and just use it in the transfer function. The user is provided with an option to make a choice regarding the calculation of $gen[s]$ and $kill[s]$ sets. Our evaluation shows that the analysis runs faster if the $gen[s]$ and $kill[s]$ sets are calculated initially rather than in the transfer function.

4 Evaluation

We assume, and do not evaluate, that data-flow analysis implementers might make errors when determining may and must sets within the specification of a transfer function, and therefore automating may/must determination is beneficial.

We evaluate the DFAGen prototype in terms of the size of the specification and predefined set specifications versus an analysis implementation in OpenAnalysis. We also compare the performance of automatically generated analysis implementations against manual implementations of liveness and reaching definitions. Our results are presented in Tables 4 and 5.

In Table 4 we compare the source lines of code of a manual and automatically generated data-flow analysis implementation. The ”Specification LOC” column shows the number of lines of code in a DFAGen specification file. A compiler writer developing the data-flow analysis would only need to write these seven lines of code. The column Pre-defined set LOC refers to how many lines of C++ code are used to specify the ‘def’ and ‘use’ predefined set structures. Since many analyses will only use predefined sets included with DFAGen, and predefined sets can be shared across multiple analyses, we believe predefined set LOC will not play a large role in most analysis specifications.

Table 5 shows the time to execute manual implemented liveness and reaching definitions analysis compared to the execution time of the automatically generated implementation. The benchmarks come from the 2006 SPEC suite. The number of lines of code analyzed is presented in the “Benchmark SLOC” column. Currently, generated analyses take longer to execute than manual implementations. However, we believe that incorporating some simple optimizations into the code generation phase of the DFAGen Tool, particularly applied to the generation of pre-defined sets, will lead to near-equivalent times as the manual-written versions.

All evaluations were done on an Intel(R) Pentium(R) 4 CPU with 2.40 GHz and a cache size of 512 MB.

5 Related Work

Guyer and Lin [6, 7] present a data-flow analysis framework and annotation language for specifying domain-specific analyses that can accurately summarize the effect of library function calls with the help of library writer annotations. Their system defines a set of data-flow types including container types such as `set-of<T>`. Their system also includes a declarative language for specifying the domain-specific transfer functions and side-effect information for calls to library routines. They enable pessimistic versus optimistic descriptions of data-flow set types, but that only de-

Analysis	Manual LOC	Automatic LOC	Specification LOC	Predefined set LOC
Liveness	394	798	7	98
Reaching Definitions	402	433	7	98

Table 4. Lines of code in manual and DFAGen generated analyses

Benchmark	Benchmark SLOC	Liveness manual time	Liveness automatic time	Reaching defs manual time	Reaching defs automatic time
470.lbm	904	0.27	0.39	0.32	0.53
429.mcf	1,574	0.54	0.71	0.62	0.89
462.libquantum	2,605	0.99	1.28	0.76	1.14
401.bzip2	5,731	11.81	13.06	43.07	52.68
458.sjeng	10,544	8.6	9.74	12.38	17.27
456.hmmer	20,658	14.41	17.95	17.02	25.88

Table 5. Evaluations with SPEC C benchmarks

termines the meet operator as being intersection or union. Using may versus must information in the transfer function appears to still be the responsibility of the tool user.

The Program Analyzer Generator, PAG [2, 9], provides a language for specifying general lattices and transfer functions. The possible data types for the data-flow set is much more expansive than what can be expressed in this initial prototype of DFAgen, therefore PAG is capable of expressing data-flow analyses other than those considered to be bit-vector analyses. PAG users express transfer functions with a fully functional language called FULA. This approach provides more flexibility in terms of specifying the transfer function when compared to the limited set builder notation provided by DFAgen. The main difference however is that in PAG a user must determine how transfer functions will be affected by pointer aliasing and side-effects. DFAgen on the other hand seeks to automate this difficulty.

The Sharlit tool [12] for building data-flow analyses focuses on enabling modularity and extensibility while automatically providing performance improving techniques to the data-flow analysis implementation, specifically path simplification. They do not discuss how aliasing and side-effects are handled. This is probably due to the fact that in a quad-based IR it is possible to perform dataflow analysis on the "register" values only and assume any memory accesses might conflict and/or be modified by any function calls.

Zeng *et al.* [13] have developed a domain-specific language for generating data-flow analyzers. The authors have developed a data-flow analyzer generator that synthesizes data-flow analysis phases for Microsoft's Phoenix compiler framework. The authors focus on intra-procedural analysis, similar to DFAGen. Similarly in DFAGen the user can specify the code for the pre-defined sets. However, AG (Analyzer Generator) does not automatically determine may and must usage of those sets. Also, analysis specification is still imperative versus declarative.

6 Conclusions

Implementing data-flow analysis even within the context of a data-flow analysis generator is complicated by the need to handle issues such as may and must pointer, side-effect, and aggregate information within the transfer function implementation. DFAGen is a data-flow analysis generator tool that given on the order of tens of lines of specification can generate all of the necessary implementation details. The presented tool depends on the availability of code for the generation of may and must versions of sets such as definition and use sets, but we present techniques that enable the tool to infer when may and must versions of the pre-defined sets are needed. Future work includes extending DFAGen to non-separable data-flow analyses and including a tuple data type within the specification language. It would be possible to extend DFAGen to allow for the specification and generation of set-based analyses by enabling the GEN and KILL set specifications to be parameterized by the incoming/outgoing set. Set-based analyses are necessary in our technique for automatic may/must determination.

Acknowledgment

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under award #ER25724. We would like to thank Paul Hovland and Amer Diwan for their comments and suggestions with regard to this paper.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, second edition*. Pearson Addison Wesley, 2007.

[2] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, pages 33–50, 1995.

[3] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java, second edition*. Cambridge University Press, 2002.

[4] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier, 2004.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[6] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *2nd Conference on Domain Specific Languages*, October 1999.

[7] S. Z. Guyer and C. Lin. Optimizing the use of high performance software libraries. *Lecture Notes in Computer Science*, 2017, 2001.

[8] G. A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[9] F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[10] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM Press.

[11] M. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *Proceedings of The Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 5-6 2005.

[12] S. W. Tjiang and J. L. Hennessy. Sharlit–a tool for building optimizers. In *The ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[13] J. Zeng, C. Mitchell, and S. A. Edwards. A domain-specific language for generating dataflow analyzers. *Electronic Notes in Theoretical Computer Science*, 164(2):103–119, 2006.